In the above code, we have created an interface named `IEncryptionAlgorithm`, which will abstract the basic properties that every concrete algorithm needs to implement. This interface is a kind of a contract which the implementation classes need to follow for writing custom encryption methods. We have:

- `Password`: the password/key for each encryption algorithm
- `RawInput`: a byte array which will hold the data that needs to be encrypted
- `Salt`: salt is needed to make sure that our encryption code is harder to crack
- `Keysize`: the bigger the key size, the tougher it is to break the encryption
- `Encrypt()` method: returns the data as a byte array after encrypting it
- `Decrypt()` method: decrypts already-encrypted data back to the original data
- `CheckPassword()`: for demonstration purposes, we will be storing the password inside the encrypted data instead of some external location. So this method will be used before the actual decryption to check if the password entered by the user is correct or not.

Step 2: Create an Implementation

We first created an `XOR` based `XOREncryption` class, which implemented this interface:

```
public class XOREncryption : IEncryptionAlgorithm
```

We will implement the encryption and decryption methods along with the properties; for detailed code refer the code bundle. Here is the trimmed -0 down version:

```
public byte[] Encrypt()
{
    byte[] encryptedBytes = new byte[_rawInput.Length];
    byte[] keyBytes = ASCIIEncoding.ASCII.GetBytes(_key);
    //hard coded salt value
    _salt = new byte[] {0x11, 0x78, 0x22, 0xFF, 0xAC, 0x5C,
                        0x78, 0x4E, 0x7D, 0x45, 0xEF, 0xF1};
         //rest of the code goes here
}
```

We need to complete this class with other methods and properties defined by the interface (see the source code provided in the code bundle).

Step 3: Create another Implementation

Because we want to switch between multiple algorithmic implementations at runtime, we will create one more encryption algorithm implementation so that we can see the plug-n-play design in action:

```
public class RijndaelEncryption : IEncryptionAlgorithm
{
       //implement IEncryptionAlgorithm properties and methods
}
```

Now we have two implementations ready, and the question becomes how we dynamically instantiate one of these in the GUI.

Step 4: Create a Factory Class

For that, we need to use the Factory design pattern and create a Factory class as follows:

```
public sealed class AlgorithmFactory
{
    private AlgorithmFactory()
    { }

    public static IEncryptionAlgorithm GetSpecifiedAlgorithm()
    {
        string algoType = System.Configuration.
                        ConfigurationSettings.AppSettings["algo"];
        IEncryptionAlgorithm algoInstance;
        if (string.IsNullOrEmpty(algoType))
        {
            goInstance = Activator.CreateInstance(Type.
                 GetType("NeekProtect.XOREncryption,NeekProtect"))
                 as IEncryptionAlgorithm;
        }
        else
        {
            algoInstance = Activator.CreateInstance(Type.
                       GetType(algoType)) as IEncryptionAlgorithm;
        }
         return algoInstance;
    }
}
```